MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 2007

Recitation 8 — 9/28/2007 Solutions
Symbols and Quote

Scheme

1. **Special Forms**

   (a) *quote* - `(quote` *expr*`)`
   Returns whatever the reader built for *expr*.

   (b) '*thing* - syntactic sugar for `(quote thing)`.

2. **Procedures**

   (a) `(eq?` *v1 v2*`)` - returns true if *v1* and *v2* are bitwise identical. "Works on" symbols, booleans, and pairs. Doesn't "work on" numbers and strings.

   (b) `(eqv?` *v1 v2*`)` - like `eq?` except it "works on" numbers as well.

   (c) `(equal?` *v1 v2*`)` - return true if *v1* and *v2* print out the same. "Works on" almost everything.

Problems

1. **Evaluation** - give printed value, assuming `x` is bound to 5.

   (a) `'3`→ 3
   (b) `'x`→ `x`
   (c) `''x`→ `(quote x)`
   (d) `(quote (3 4))`→ `(3 4)`
   (e) `('+ 3 4)`→ error: symbol `+` is not a procedure
   (f) `(if '(= x 0) 7 8)`→ `7`
   (g) `(eq? 'x 'X)`→ #f (though depends on Scheme version)
   (h) `(eq? (list 1 2) (list 1 2))`→ #f
   (i) `(equal? (list 1 2) (list 1 2))`→ #t
   (j) `(let ((a (list 1 2))) (eq? a a))`→ #t

Boolean Formulas

A boolean formula is a formula containing boolean operations and boolean variables. A boolean variable is either `true` or `false`. `and`, `or`, and `not` are all boolean operations. For the purposes of this problem, `and` and `or` will be defined to take exactly two inputs.

Example formulas:

```
a
(not b)
(or b (not c))
(and (not a) (not c))
(not (or (not a) c))
(and (or a (not b)) (or (not a) c))
```

Some useful procedures:

```
(define (variable? exp)
  (symbol? exp))
(define (make-variable var)
  var)
(define (variable-name exp)
  exp)

(define (or? exp)
  (and (pair? exp) (eq? (car exp) 'or)))
(define (make-or exp1 exp2)
  (list 'or exp1 exp2))
(define (or-first exp)
  (cadr exp))
(define (or-second exp)
  (caddr exp))

(define (and? exp)
  (and (pair? exp) (eq? (car exp) 'and)))
(define (make-and exp1 exp2)
  (list 'and exp1 exp2))
(define (and-first exp)
  (cadr exp))
(define (and-second exp)
  (caddr exp))
```

4. Write selectors, constructor, and predicate for `not`

```
(define (not? exp)
  (and (pair? exp) (eq? (car exp) 'not)))
(define (make-not exp)
  (list 'not exp))
(define (not-operand exp)
  (cadr exp))
```

5. Given a boolean expression and a set of variable assignments, evaluate the expression to decide whether the result is `#t` or `#f`. Assume that you have a procedure `(variable-value name environment)`, which takes a variable name and list of values and returns the value assigned to the variable, if a binding for it exists, or throws an error if no binding is found.

```
(define (eval-boolean exp env)
  (cond ((variable? exp)
         (variable-value (variable-name exp) env))
        ((not? exp)
         (if (eval-boolean (not-operand exp) env)
             #f
             #t))
        ((or? exp)
         (if (eval-boolean (or-first exp) env)
             #t
             (eval-boolean (or-second exp) env)))
        ((and? exp)
         (if (eval-boolean (and-first exp) env)
             (eval-boolean (and-second exp) env)
             #f))
        (else (error "unknown exp" exp))))
```

6. The evaluator as described so far only allows expressions to be either boolean operators or variable values. Extend the operator so that expressions can include literal booleans as well, so that evaluating expressions such as `(and #t #f)` work.

   Add the following expression to the set of conditions:

```
((boolean? exp) (boolean-value exp)
```

   `boolean?` can stay the same as the Scheme built-in of the same name, but `boolean-value` should be defined as:
   `(define (boolean-value exp) exp)`